Mikael Lindlöf

Caching Proxy Development

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

8 April 2015

Metropolia

Author	Mikael Lindlöf
Title	Caching proxy development
Number of Pages	54 pages
Date	8 April 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Production
Instructor(s)	Santeri Salminen, Project Manager Olli Hämäläinen, Senior Lecturer

Methics Oy has developed a new software product called Kiuru Caching Proxy (CP). It will be offered as a part of Methics' Kiuru MSSP software family. The purpose of a CP is to support other MSSP products. This thesis documents the purpose, development and functionality of CP.

The development was influenced by the needs and requirements of MSSP products. CP aims to offer features that are helpful in use cases that MSSP products set. MSSP's requirements had to be taken into account, for example, during cache functionality development. Cache uses database in a way that allows scaling and decentralization of CP servers.

CP provides a platform and a framework for CP services. The development was based on an existing proxy product called ASCache. Commonly useful parts of ASCache were used in CP. A service called NPC was developed along with CP.

As a result, CP provides the NPC service. More CP services are added in the future when needed.

In conclusion, CP helps Methics to develop more uncluttered MSSP products since CP services do a part of MSSP service's work. This helps with code maintainability and system administration.

Keywords	Proxy, cache, mobile signature, authentication

Tekijä Otsikko Sivumäärä Aika	Mikael Lindlöf Välimuistavan välityspalvelimen kehitys 54 sivua 8.4.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotuotanto
Ohjaaja(t)	Projektipäällikkö Santeri Salminen Lehtori Olli Hämäläinen

Methics Oy kehitti uuden ohjelmistotuotteen nimeltä Kiuru Caching Proxy (CP). Sitä tarjotaan osana Methicsin Kiuru MSSP-ohjelmistoperhettä. CP:n tehtävä on tukea muita MSSP-tuotteita. Tämä raportti dokumentoi CP:n tarkoitusta, kehitystä ja toiminnallisuutta.

Kehitystä ohjasivat MSSP-tuotteiden tarpeet ja vaatimukset. CP:ssä pyrittään tarjoamaan ominaisuuksia, jotka auttavat MSSP-tuotteiden asettamissa käyttötapauksissa. Muun muassa cache-toiminnallisuutta kehittäessä täytyi ottaa huomioon MSSP:n vaatimukset. Cache käyttää tietokantaa tavalla, joka mahdollistaa CP-palvelinten skaalaamisen ja hajauttamisen.

CP toimii alustana ja runkona CP-palveluille. Kehityksen pohjana käytettiin olemassa olevaa proxy-tuotetta nimeltä ASCache, josta otettiin CP:seen yleiskäyttöiset osat. CP:n ohella työssä kehitettiin CP-palvelu nimeltä NPC.

Projektin tuloksena CP tarjoaa NPC-palvelun. CP-palveluja lisätään tulevaisuudessa tarpeen mukaan.

CP auttaa Methicsiä kehittämään tulevaisuudessa selkeämpiä MSSP-tuotteita. CP-palvelut tekevät osan MSSP-palveluiden töistä. Tämä helpottaa sekä koodin ylläpidettävyyttä että järjestelmän ylläpitoa.

Avainsanat	Välityspalvelin, välimuisti, mobiilivarmenne, tunnistus

Contents

Abbreviations

AP	Application Provider. Entity that requests authentication from MSSP [1,9].
CP	Caching Proxy. Methics' solution to proxy requests from an MSSP to a CS.
CPP	Caching Proxy Provider. Entity that provides CP services.
CS	Content Service. Service that provides content when requested.
CSP	Content Service Provider. Entity that provides a CS service.
JSON	JavaScript Object Notation. Machine and human readable data-interchange format [2].
MSS	Mobile Signature Service. Web service that allows data signing with mobile devices [1,12].
MSSP	Mobile Signature Service Provider. Entity that provides an MSS service [1,9].
MSISDN	Mobile Station International ISDN Number. Identifies a subscription in a mobile network.
NPC	Number Portability Cache. Methics' solution for caching number portability content [3].
NPCP	Number Portability Cache Provider. Entity that provides an NPC service.

1 Introduction

Methics Oy is developing a new software product called Kiuru Caching Proxy (CP). It will be offered as a part of Methics' Kiuru Mobile Signature Service Provider (MSSP) software family. Customers that buy Kiuru MSSP are most often mobile operators.

Kiuru MSSP provides Mobile Signature Service (MSS) that allows strong user authentication with mobile devices. Any party that requests authentication from an MSSP is called an Application Provider (AP). An AP can authenticate a user by sending an authentication request to an MSSP. Information about an authenticated user is called user identity. User identity is provided in an authentication response.

During authentication the MSSP system often requests additional content about the user identity that is being authenticated. The Kiuru MSSP system has two known use cases for additional content:

- Routing authentication requests to the right MSSP during route discovery [4,16].
- Supplementing user identity in an authentication response with Additional Service information [1,39].

The party that provides content is known as a Content Service Provider (CSP). Quite often the AP can request an MSSP to supply content from a CSP. The CSP may provide any content it has about the user, for instance, name or birthday.

CP is a solution to integrate MSSPs with CSPs. CP makes content requesting easier for MSSPs by dealing with requirements and limitations of CS protocols. CP uses a cache database to optimize response times. Isolating optimization and protocol issues from the MSSP software to CP is practical. This matches Methics' view of good software design.

This thesis documents the purpose, functionality and development of CP. MSS environment sets some security and performance requirements for CP. As a platform for different proxy services CP needs to support diverse functionalities and provide different extension points.

2 Overview

2.1 MSS System

MSSPs allow strong user authentication with mobile devices. An end user can, for example, login to a web service. Mobile authentication and Finnish bank authentication TUPAS [5] are both commonly used by Finnish web services that require strong authentication.

This chapter describes a typical MSSP environment use case. In this use case a user logs into a web service that requires strong authentication and an AP authenticates the user with MSS. A Caching Proxy Provider (CPP) is an entity that provides CP services. CPPs are a part of MSSPs in the scope of this chapter.

In this use case the user provides a Mobile Station International ISDN Number (MSISDN) to a web service. An MSISDN is, simply put, a telephone number. In this scenario the web service acts as an AP. The web service requests an MSSP for authentication of the given MSISDN.

The authentication request is routed between MSSPs and sent to the MSSP that sends the request to the telecom operator's mobile network system [4,7]. Any number of MSSPs could be contacted while routing the authentication request. The mobile network system sends a request to the user's mobile device via an SMS. Figure 1 demonstrates an authentication request.

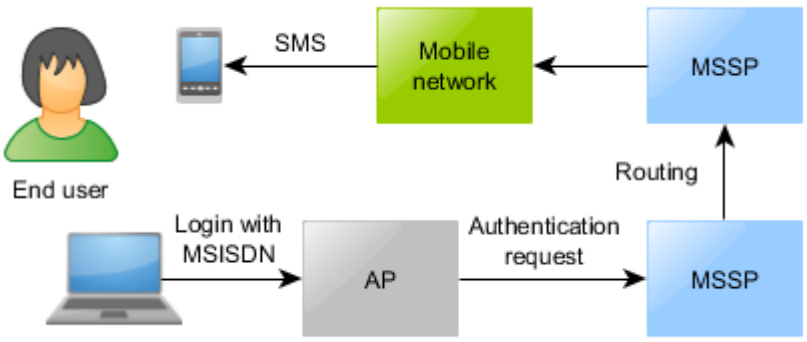

Figure 1. MSS environment, based on ETSI TS 102 204 and ETSI TS 102 207 [1;4]

A SIM card application processes the authentication request. The user accepts authentication with their mobile device and provides a PIN code to the SIM card. The SIM card makes a cryptographic signature with the user's private cryptographic key. The end user must have their SIM card and know their PIN code to produce a valid signature. This fills the requirements for two-factor authentication which is a necessary part of strong authentication.

The authentication response is sent back from the SIM card to the mobile network system and the MSSPs. The MSSPs send the authentication response back to the web service. The authentication response includes user identity. The user identity consists of the user's MSISDN, signature and Additional Service responses. The web service processes the login request based on authenticated user identity.

Kiuru MSSP servers can have different roles. One MSSP can be dedicated to authenticating and dealing with incoming authentication requests. Another can be dedicated to sending requests to a mobile network system. Requests are routed between MSSPs until they are sent to the mobile network system.

Kiuru MSSP can be used in many ways to perform different tasks. The MSS environment always functions similarly to the environment demonstrated in this chapter. MSSPs always take a request, roam it and send it to a mobile network system.

In strong authentication there must be strong evidence and validation that the end user is the person they claim to be. Transaction of strong authentication must be verifiable at a later time.

Kiuru MSSP systems also provide lighter authentication when an AP requests for it. Often it is enough for the AP that an end user is, with high probability, the same person as the last time. End users could benefit by having a lighter registration process, faster login process and easier login (possibly without PIN code). This is, in many cases, stronger and easier than password authentication.

2.2 CP Environment

Kiuru MSSP products need content from different CSPs. Each CP service provides an interface for an MSSP to retrieve content from a CSP. CP works as a host platform for services.

A CP service has two interfaces. A CS Proxy protocol is a protocol that a CP service provides for an MSSP. A CS protocol is a protocol that a CSP provides for a CP service. These protocols are shown in figure 2.

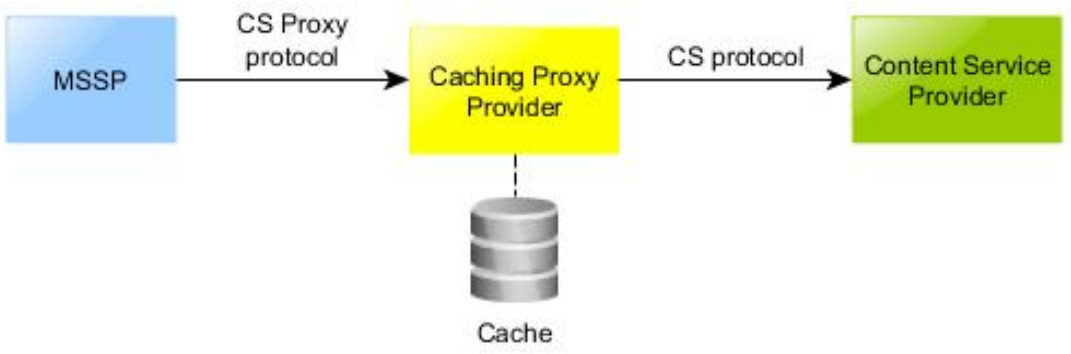

Figure 2. Kiuru Caching Proxy environment

Any Kiuru MSSP server can make a request to a CPP. Different MSSP servers may use different CP services. All CPPs are packaged with the same services. Availability of a service in a CPP is modified by configuration and licensing.

A CSP may provide features that MSSPs do not need. Additional features often make a CS protocol more complex. A CP service should provide a simpler CS Proxy protocol by concealing complications that are unnecessary for an MSSP's use. However, CS Proxy protocols should be extensible because missing features may be required in the future.

CP helps to simplify MSSP functionality in the following ways:

- CS protocols are often different. CS Proxy protocols can work similarly and MSSPs can reuse logic between protocols.
- A CS protocol may have complexities and limitations. CS Proxy protocols can be customized to needs of MSSP servers.
- CP can make optimizations such as caching.

CS Proxy protocols are usually completely under Methics' control. This is because the purpose of CP is to provide services for Methics MSSP products. However, a customer may want to use another CS Proxy protocol. This could be the case when the customer wants to, for example, use a CP service from their own products.

2.3 Caching Services

CSPs are often slow and have limited resources. CP can avoid contacting CSPs by caching routing content. This allows CP to provide responses faster.

CP provides a transparent caching mechanism. CP services route content request through transparent caching. Content is returned if it is in the cache. If content is not cached then the request is forwarded and the response is cached.

Initially the request is forwarded to a component that retrieves content from the CSP. Later content is returned from the cache. Figure 3 shows an example of using transparent caching.

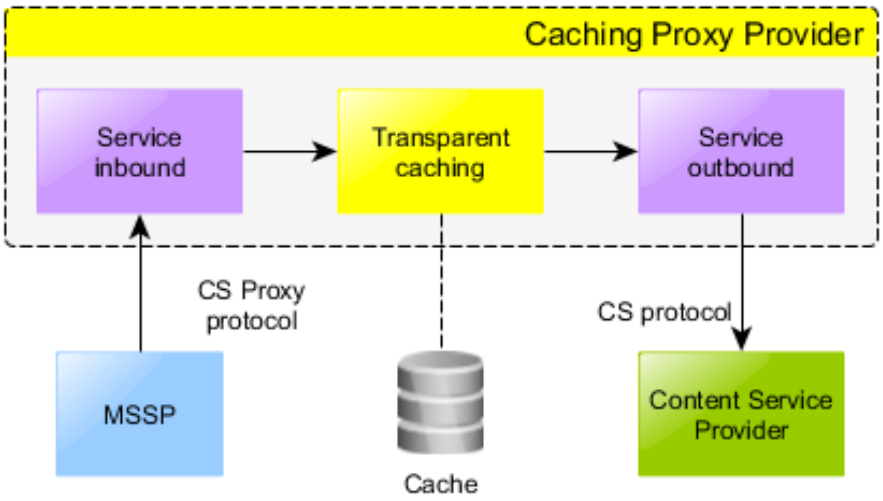

Figure 3. Transparent caching

Caching content makes requests faster since content is readily available. Additionally CPPs can be located near MSSPs to reduce connection latency. Often CSPs have no caching support of their own. Caching is often needed for a CSP to be able to work effectively with the load that MSSPs generate.

Deciding if cached content needs to be refreshed is the largest issue with caching. Some services might have a solution to detect deprecated cache entries and some might not.

Additional Services usually cannot use caching. There is usually no way to detect if cached Additional Services content is deprecated.

Routing Content Caching

The telecommunications operator of an end user is responsible for communicating with their mobile network system and SIM application. An authentication request is routed to an MSSP that belongs to the end user's operator. MSSP systems of different parties are often connected. Connection to all operators that support the MSS is necessary so that authentication requests can reach end users from all operators.

Request routing often requires additional content instantaneously so that routing can continue processing. For example, external content may be required to resolve the current telecommunications operator of an MSISDN.

MSSP routing either succeeds or fails with a routing failure. Used additional content comes under scrutiny when a routing failure is received. Routing failure can be used as a way to detect deprecated cache entries. After a routing failure the MSSP can request refreshed content from a CPP.

2.4 Asynchronous Services

A user identity can be supplemented with Additional Services. An MSSP can declare Additional Services. Then, an AP can request supported Additional Services in an authentication request. For example, the MSSP could support Additional Services for requesting name, timestamp, birthday and social security number.

The response time of the whole authentication process depends on several factors. The most notable factors are mobile network speed and user input speed.

Additional Service response times usually must not (significantly) affect authentication time. A client connecting to CSP must potentially wait for a long time for the connection to end. Synchronous processing would often require the MSSP to wait for a long time for the CSP connection to end.

An asynchronous protocol allows the MSSP to perform other processing while content is retrieved. CS protocols often do not allow asynchronous requesting. A CP service can implement an asynchronous CS Proxy protocol.

Routing does not benefit from asynchronous service. The routing process needs routing content instantaneously to continue processing.

Additional Service Polling

Polling is directed to an asynchronous CP service. The CP service does not provide content in an initial response. The MSSP sends the first request when it receives an authentication request. When caching is not used the first request is an initial request.

The requested content is not cached and is not being asynchronously requested before the initial request. The initial request has the following steps:

1. The MSSP requests the CP service for content when receiving an authentication request.
2. The content is not in the cache so the CP service acknowledges the request to MSSP.
3. The CP service creates a separate thread where it requests the content from the CSP.
4. The CSP responds with the content and the CP service stores it in the cache.

The initial request is demonstrated in figure 4. The numbers in the figure match numbering in the above list.

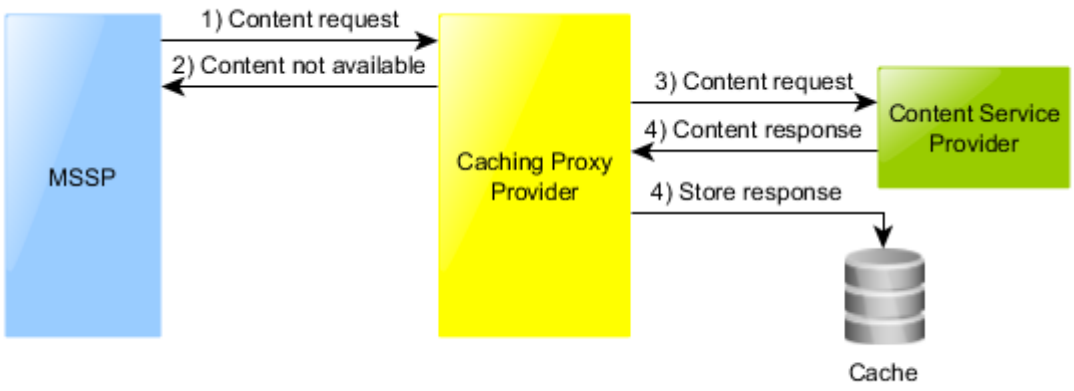

Figure 4. Initial asynchronous content request

CSP content is stored in the cache when the asynchronous process of the initial request has finished. The MSSP makes a second request when responding to the authentication request. Normally the initial request has finished by this time.

A content request is called the final request when the content is stored in the cache. The final request has the following steps:

1. The MSSP requests the CP service for the content for the second time when responding to the authentication request.
2. The CP service responds with the cached content.

The final request is demonstrated in figure 5. Numbers in the figure match numbering in the above list.

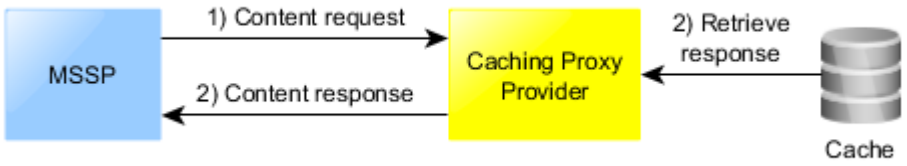

Figure 5. The final asynchronous content request

Before content is cached, the CP service always responds with an acknowledgement. When content is cached, the CP service always responds with the content.

Second Request Before CSP Response

The MSSP makes the second request to the CP service when responding to the authentication request. The second request qualifies as the final request if content is cached. In case that content is not cached the CP service acknowledges the second request like the initial request.

The MSSP needs to be fast and reliably deliver an authentication response. The MSSP gives the CP service a considerable amount time to retrieve content between the initial and the second CP request. End user authentication, which MSSP performs between the CP requests, speculatively takes minimum of 10 to 20 seconds.

Usually the MSSP cannot keep waiting for Additional Services content. The MSSP would prolong authentication time if it waited for the content after the second request.

Usually, in this case, the MSSP does not provide the Additional Service that requires the content.

Status Request

The status of the authentication can be requested from the MSSP [1,18]. The MSSP responds with a status response. The status response is an acknowledgement or, if authentication is complete, the outcome of the authentication. The CP service is not contacted when the MSSP acknowledges a status request.

The status response contains the outcome of the authentication and is similar to an authentication response. The Kiuru MSSP servers do not store Additional Service responses. Therefore, for the status response, MSSPs need to request the content from the CP service again. The CP service either acknowledges the request or responds with the content if the content is stored.

A service that uses the cache only for asynchronous purposes must clean old content from time to time. Keeping old temporary content would be a waste of database space.

Cleaning the content while status requests are made would suddenly make the content unavailable. The CP service would start retrieving the content from the CSP again when the content is requested from the CP service. Repeated status requests could get responses with an Additional Service that is sometimes unavailable.

CP services with asynchronous content should not have a too short lifetime. The lifetime should be configured to be longer than the duration for which the MSSP can make repeated requests. This configuration prevents content being unavailable or changing.

The lifetime of a cached response can be controlled with CP configuration. The lifetime may not be configurable in all CSPs. Caching configurability is an additional benefit of using CP for temporarily caching asynchronous content.

Polling Efficiency

Asynchronous CP services respond to MSSPs with or without content depending on whether requested content is available. The MSSPs know that the asynchronous CP services will instantly respond and close connections. Also the MSSPs can keep requesting for content until it is received. This process is called polling. The MSSPs benefit by being able to make decisions based on content availability and to keep processing instead of waiting for a connection to end.

Content polling is considered inefficient in most use cases. This is because polling generates a considerable amount of traffic between a client and a server. The frequency in which the client polls the server defines how fast the client receives content once it is available (cached in the CPP). Push mechanisms such as server callback, Long-Polling and WebSockets could solve this issue [6].

MSSP Additional Services do not depend on content availability. They rather fail to deliver Additional Service than keep waiting for content. Simple polling allows the MSSP to request for content and always get a response instantly.

Cache Usage

Caching means increasing performance by storing responses. Temporarily storing data for asynchronous polling is not caching. Nevertheless, the caching database and logic can be used for temporarily storing data. The two use cases for the cache are not mutually exclusive either; the cache can be used just for caching, just for asynchronous storage or both.

A service decides if it uses the cache for caching or temporary storage. Services configure the cache differently depending on whether they need caching or an asynchronous protocol.

2.5 Number Portability Cache

Number Portability Cache (NPC) is used to cache number portability content. The NPC content is used as a part of MSSP roaming. NPC will be implemented as a service in CP. An entity that provides the NPC service is called a Number Portability Cache Provider (NPCP).

Telecommunications operators have their allocated prefixed set of MSISDNs. A mobile user receives an MSISDN when they sign up with an operator. Telecommunication is routed to an operator by MSISDN's prefix that is allocated to the operator.

When mobile users change their operator in Finland, it is possible to preserve their mobile number with a number portability service called Numpac [7]. Ported MSISDNs are stored in the service with information of their current operator.

Routing ported MSISDNs requires consulting Numpac. All Finnish telecommunications use the Numpac service and Numpac has certain limitations in its capacity. Therefore, Numpac usage has strict rules to prevent heavy usage. [7, 3.]

MSSPs can roam requests based on an MSISDN [4,16]. The MSSP requests number portability content from an NPCP instead of requesting it directly from Numpac. The NPCP serves cached content if the content is cached or forwards the request to Numpac if not. Numpac responses are cached and served from the cache later.

The MSSP returns a routing error when the MSISDN's operator is not found [4,24]. The routing error is an indication that cached content in the NPCP may be expired. In this situation the MSSP asks for the NPCP to refresh cached content for the MSISDN. The MSSP can try routing again with refreshed content.

The NPC content is required instantly for the MSSP to be able to continue processing. Asynchronous processing would require the MSSP to continuously and unnecessarily poll a CPP. Therefore, it is better for the NPCP to work synchronously.

Routing mostly works even when Numpac is offline. Some ported numbers will work since they are cached in the NPCP. When the NPCP cannot deliver content, the MSSP reverts to configured MSISDN prefixes.

There is also a configurable lifetime for the NPC content so that the content is automatically refreshed. The NPC content lifetime is typically one year.

Numpac provides a level of reliability with content. Having requested an MSISDN stored in its database results in a high level of reliability. Alternatively Numpac provides a response with a lower level of reliability using operator prefixes. This resembles the way Methics always wants to handle routing content. CPPs should provide any available information and information's metadata to MSSPs and let the MSSPs make routing decisions.

2.6 CP Product Requirements

MSSP Operation

CP exists to add convenience for MSSPs. Any operation that a CPP does could be also performed in MSSPs. The CPP should not make decisions for MSSPs but only provide data received from CSPs. The CPP's mission is to provide easy but powerful interfaces to content services.

Strong authentication sets high security requirements for CP. CP is part of the MSS system and has the same security requirements. Other use cases may set lower security requirements for the MSS system.

CP must support asynchronous proxy services. CSPs are slow so MSSPs gain performance by running CSP connections asynchronously with MSSP processing.

Invalid Additional Services content must not be provided. Providing invalid content in Additional Services can cause a breach in authentication. For example, providing a

wrong social security number can cause authentication as someone else. Failing to provide Additional Service content is better than to provide invalid content.

CP must support caching of routing content. MSSPs need to wait for CSPs, which can be slow, since routing content is often requested synchronously. Caching can speed up future requests. CSPs also often have poor optimizations and MSSP services may generate too much load. Caching helps by reducing the amount of CSP requests.

Providing invalid content for routing can deny service from end users. MSSPs must be able to refresh content on CP in case of a routing error. CP should provide all available content with their metadata and let MSSPs make routing decisions.

Service Deployment

CP and services are written in Java 8. CP is run on Kiuru server platform. Methics servers run on Kiuru platform so that products are easy to maintain and have similar user experience. Kiuru platform works on top of Apache Tomcat web server. Kiuru platform deals with server installation, configuration and server status.

Scalability is important in the MSSP system. For example, millions of people could be using the MSS to buy a ticket for a morning train at the same hour. Performance should increase drastically when adding new servers. CP is required to also take scalable design and heavy usage to consideration.

The MSS must work even when one geological service fails. Support for geographic redundancy must be possible when using CP services.

CP Development

Ideally content caching should be fully transparent. Removing and adding caching to a service should be easy and modular.

Different ways of providing content must be supported. Development of modular content sources, similar to the modular caching feature, must be supported. Another way of providing content could be, for example, asking other proxies for content.

For business reasons it is not always possible to use Methics' own protocol between MSSPs and CPPs. CP cannot place many restrictions on CS Proxy protocols and CS protocols. CP can expect CS Proxy protocols to be based on the HTTP protocol.

The functionality of NPC will be moved to CP. MSSPs will use the NPC service that CP provides.

3 Development

3.1 Project Starting Point

Derivation from NPC

Methics has an old NPC product that works as a standalone server [3]. Replacing NPC server by moving the Numpac caching functionality to CP was an object of this project.

The starting point of this project was to identify the parts of NPC server that are NPC specific. The functionality that is not NPC specific could be shared between different services. All services could use the shared functionality.

For business reasons another service called ASCache was developed before starting this project. ASCache was a better starting point than NPC. ASCache had support for the Kiuru platform and included much of the design that CP was going for. On the other hand, NPC provided more limited infrastructure.

The idea of using NPC as a starting point was abandoned. Apart from some NPC specific parts, mainly the Numpac client, the NPC server had no contribution to the CP project.

ASCache Development

ASCache required caching capabilities similar to CP. MSSP can request an ASCache server for Additional Service content. The ASCache server retrieves content asynchronously while the MSSP is retrieving user input.

ASCache is used to request SubscriberInfo content from a customer's own system. The customer system only provides a synchronous interface. The request from the MSSP includes a set of variable names that ASCache will request for in a SubscriberInfo request.

ASCache provides a special ASCache protocol for the MSSP. The ASCache protocol has strict specifications about HTTP response status codes.

One aim in ASCache development was designing the ASCache code to be reusable with CP. A chapter on ASCache design documentation called “Service and Cache” was aimed to guide the design of ASCache for reusability in CP [8,10]. The chapter describes how the REST servlet, ASCache service, ASCache Workers, the cache and Fetchers work together.

ASCache implementation does not actually have a clear separation of the cache, Fetchers and the proxy components. ASCache only contains the ASCache service so modularity does not provide value to ASCache.

CP is now based on ASCache instead of NPC. ASCache was created with modular service and Fetcher design in mind. It was helpful even when modularity was not actually fully implemented.

The NPC service is implemented into CP as planned. The ASCache service is not implemented in CP. The ASCache service is kept separate from CP since the ASCache service will not require any future development. Doing changes to ASCache could break the ASCache service, and the service works as it is.

Product Changes

Before starting the CP project Methics had the products shown in figure 6. ASCache was already developed before the CP project started.

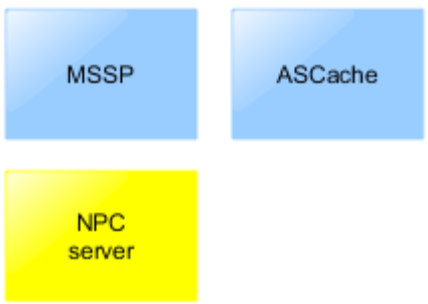

Figure 6. Project starting point

After CP was ready for production usage Methics had the products shown in figure 7. NPC server was replaced with CP that hosts the NPC service.

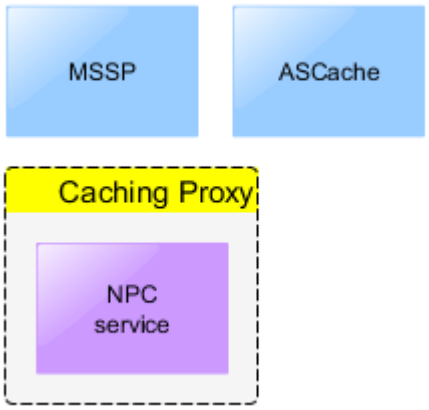

Figure 7. Project outcome

To summarize, ASCache was used as the starting point in CP and NPC service development. NPC server is no more needed and it is removed. NPC server's Numpac client is used in the NPC service. MSSP will now use the NPC service instead of NPC server.

3.2 Service Framework

Proxy Service Design

CP is a framework for proxy services. A service provides an interface for MSSPs and a client for a CS. There can be multiple services in a CPP.

Proxy Servlet

Proxy servlet is the component that receives all incoming requests. Proxy servlet takes HTTP GET and POST requests and routes them to services using the request URL. For example, a request to the URL http://localhost:9000/proxy/service/npc/get is routed in the following way:

- Tomcat hosting a CPP receives requests coming to http://localhost:9000.
- Proxy servlet receives requests coming to /proxy/service.
- Proxy servlet routes the request to service using the next part of the request URL. The next part in the example is /npc.

Later in processing CP services often uses the next part of the request URL to route the request to a component called Worker. Figure 8 demonstrates the routing of a CP request.

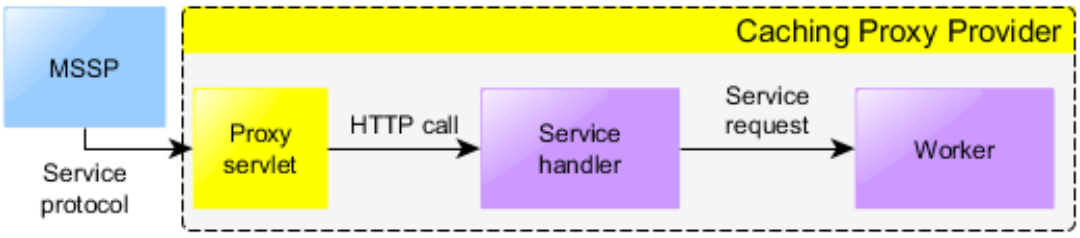

Figure 8. Request routing

While all different services (such as NPC) could have separate servlets, a single common servlet is preferred. Using a single servlet has the following consequences:

- Services are more lightweight since servlets take resources.
- Adding a new service is easier.
- Less Tomcat configuration.
    - Initialization and configuration of services is controlled by Proxy servlet.
    - Request routing is controlled by Proxy servlet.
    - Service availability is only controlled by service configuration and licensing.
    - Service URLs are less manageable.
- Proxy servlet takes part of the service URL. Proxy servlet URL is “proxy/”.
    - API call URLs of all services are under “proxy/service/”.
    - Documentation of all services is under “proxy/docs/”.

In addition to routing HTTP requests, Proxy servlet also simplifies them. Proxy servlet handles different formats of HTTP requests and unifies all requests to a common and easily usable format. Proxy servlet does not enforce or hide any data since it also passes the original request to Service handler.

Service Handlers

Service handlers receive HTTP requests that come from Proxy servlet. Service handlers deal with the HTTP layer. Service handlers can perform the following tasks:

- Validation of HTTP request (for example URL and headers).
- Validation of request data format (for example requiring JSON type data).
- Routing request forward.
- Authentication and authorization of user credentials.
- Construction of HTTP response.

Service handlers route messages to Workers based on some criteria. Usually the criterion is the last part of the request URL. For example, the NPC service handler would route a request with the URL /npc/get to the "get" Worker of the NPC service. In the figure 8 a Service handler routes a request to a Worker.

Most services use the Default service handler. The Default service handler helps when the common protocol format is used for a CP service. The default request format is used when a different protocol format is not required. The Default service handler is not used when service uses different routing method, handles HTTP differently or uses something else instead of JSON type messages.

Worker structure and JSON handling of the Default service handler is derived from another Kiuru product. In the other product there is no need for different types of services (as they are called in the context of CP). Most of the code is rewritten for CP.

Worker

Workers are service API calls. For example, a request with the URL /npc/get would be routed to the "get" Worker of the NPC service.

A Worker interprets and validates request data (for example JSON type data). After that, the worker performs the requested task. When the given task is data retrieval, the Worker will use Fetchers to fetch data that originates from a CSP.

The request parameters are given to the selected Worker when available. Some Workers may not need request parameters and some may return an error when the request parameters are omitted. For example, a clean Worker might not need any parameters. The clean Worker could use configured lifetime to delete old cache entries.

Deriving from REST API

Worker design is taken from another Kiuru product called REST API. REST API was a natural basis for the ASCache project since it was the only Kiuru product that handles JSON messages.

Requirements of REST API and ASCache are different from CP. Both have only one service but in REST API Workers can be dynamically added. Clearly separating different services is more important than dynamic addition of Workers in CP. CP services have clearly different purposes and separate states. In contrast, the REST API service is completely stateless.

3.3 Content Fetcher Interface

Workers request content from Content Fetchers. Content Fetcher is an interface that content sources implement. The Content Fetcher interface is notably implemented by Cache Fetcher. The Content Fetcher interface provides a routable “Get content” procedure.

The NPC “get” Worker, for example, uses two Content Fetchers. Cache Fetcher provides transparent caching for NPC. Numpac Fetcher is a CS specific Fetcher that requests content from Numpac.

Routing a fetch request through multiple Content Fetchers is called chaining. Chained Fetchers can either provide content or continue to the next Fetcher in the chain. The purpose of CS Fetcher is to contact a CSP if other Fetchers in the chain cannot provide a response. CS Fetcher is always the last Fetcher in the chain.

In NPC a fetch request is first given to Cache Fetcher. The provided Fetcher chain is "numpacFetcher". In case of a cache miss, Cache Fetcher routes the content Fetch request to Numpac Fetcher. This case is demonstrated in figure 9.

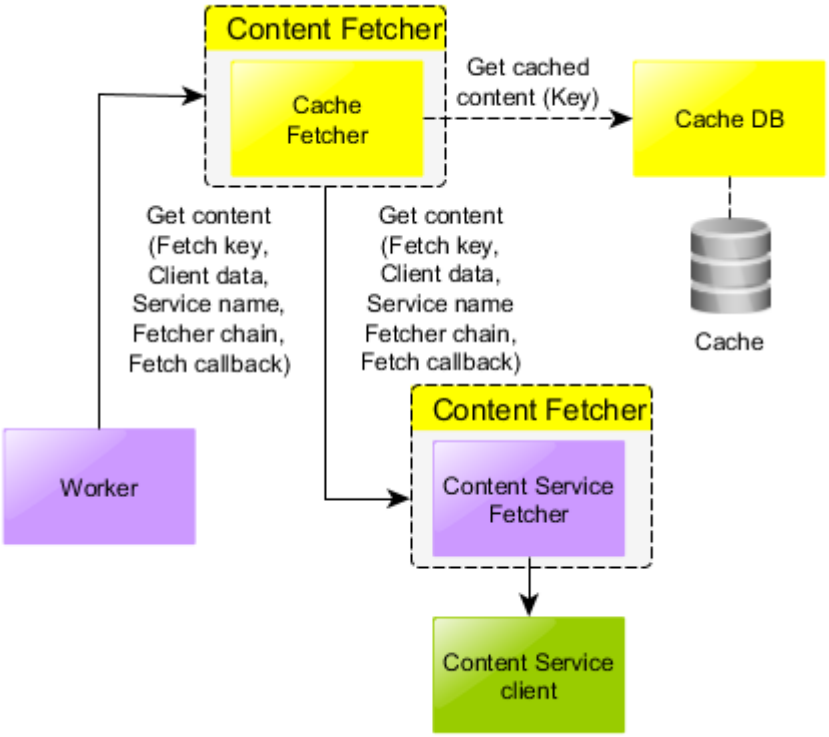

Figure 9. Content retrieving

"Get content" has the following parameters:

- "Fetch key" that identifies the requested data or transaction.
- "Client data" that CS Fetcher uses to make a request.
- "Service name" defines the service that made this call, for example "npc".
- "Fetcher chain" defines Content Fetchers that this request will be routed to, for example "someFetcher,numpacFetcher".

- “Fetch callback” for callback processes.

“Fetch key” Parameter

Workers interpret or derive a “Fetch key” from request parameters. A Worker forms the key depending on how its service identifies a response.

The key is usually a JSON object that contains usable data. Key data is not unmarshalled and is only used to identify “Client data”. All necessary data is given in “Client data”.

“Client data” Parameter

Workers construct “Client data” that CS Fetchers can use to make a request. This may be the service request itself, be explicitly defined in the request parameters or be derived from the request parameters.

CS Fetcher does not need to really understand the service request but it simply feeds the “Client data” to the client. Workers perform checks on the request and are aware of its semantics so it is a good place to form the “Client data”.

“Client data” includes the client operation when the client supports multiple operations. Client operations usually match Worker operations, for example the “get” Worker would fetch CSP content using a “get” CSP call.

“Fetch callback” Parameter

The typical way for a Java method to return a value would be using the return statement. Code that calls a method needs to wait for the called method to end to get the return value.

Fetchers support asynchronous processing in order to support asynchronous services. Code must be able to run a “Get content” procedure asynchronously. Waiting for the “Get content” procedure to end would be synchronous processing.

Instead the caller provides a callback object that implements the “Fetch Callback” interface. Content Fetcher calls the Response method of given “Fetch Callback” before returning.

Figure 10 shows how Cache Fetcher uses Cache Callback to store response content in asynchronous configuration. Cache Fetcher makes a new thread for asynchronous processing.

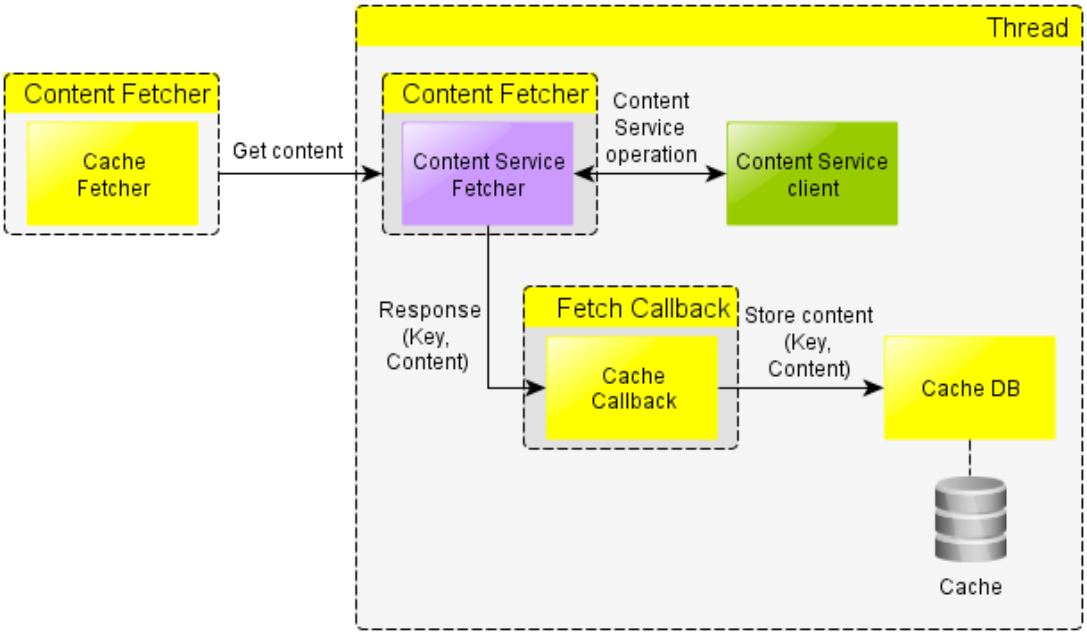

Figure 10. Asynchronous “Fetch Callback”

Content Fetcher does not always run asynchronously. It is unnecessary to create more than one new thread to allow asynchronous service. In this case a Synch Callback object can be used to temporarily store response content.

Figure 11 demonstrates providing a Synch Callback object to a Fetcher. Synch Callback stores the response key and content. Once Cache Fetcher returns, the Worker reads the response key and content from Synch Callback.

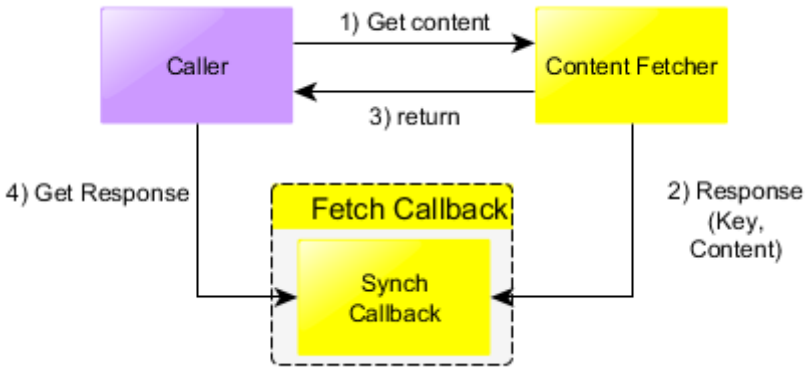

Figure 11. Synchronous "Fetch Callback"

"Fetch Callback" that Fetchers receive on "Get content" should always be run. The content will probably not always be available for Fetchers that run before a new thread is created. This is because after creating a new thread the old thread will quickly return to the Worker.

All Fetchers should provide received "Fetch Callback" to their own "Fetch Callback". In figure 12 Fetch Callback C contains all the previous callbacks recursively.

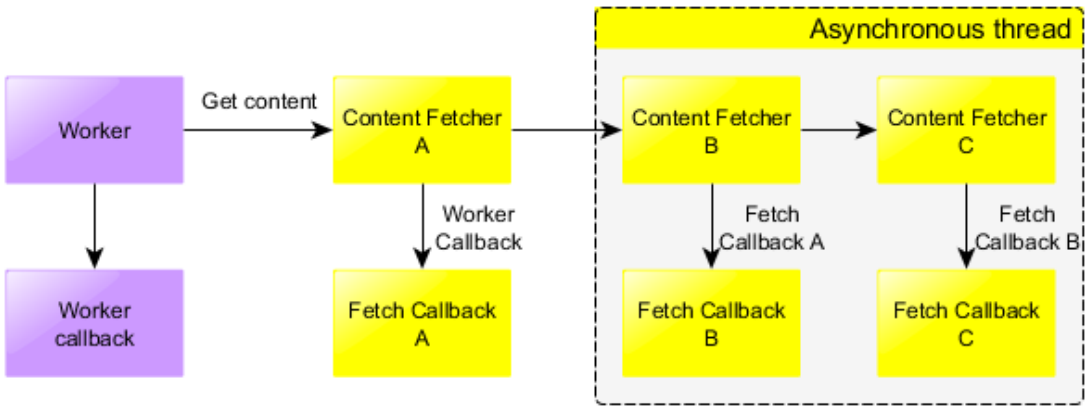

Figure 12. "Fetch Callback" creation

"Fetch Callback" should run the previous "Fetch Callback" after processing the response. The previous "Fetch Callback" is called Request Callback. In the example Fetch Callback C should run Fetch Callback B and so on. This way all Request Callbacks are called once content is retrieved.

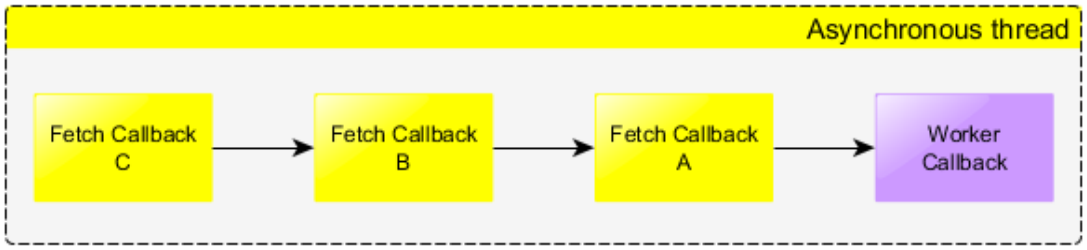

Figure 13. "Fetch Callback" objects calling Request Callbacks

Callers that use Synch Callback can only process the response if it is retrieved in the same thread. Even Synch Callback runs the previous callback since previous callbacks might handle the response asynchronously.

3.4 Fetch Response

The fetch response is stored in Response Wrapper. Response Wrapper is a JSON type object that is returned as a string. Therefore Fetchers can, for example, easily receive Response Wrapper from another server. The point of using JSON is making Fetchers more independent components. The returned value is expected to be a Response Wrapper type object.

CS Fetcher creates the Response Wrapper. CS Fetcher stores content as an HTTP message and marks HTTP Status as successful. Error HTTP Status is returned by the Worker or Fetchers in case of an error. Response Wrapper returning is shown in figure 14.

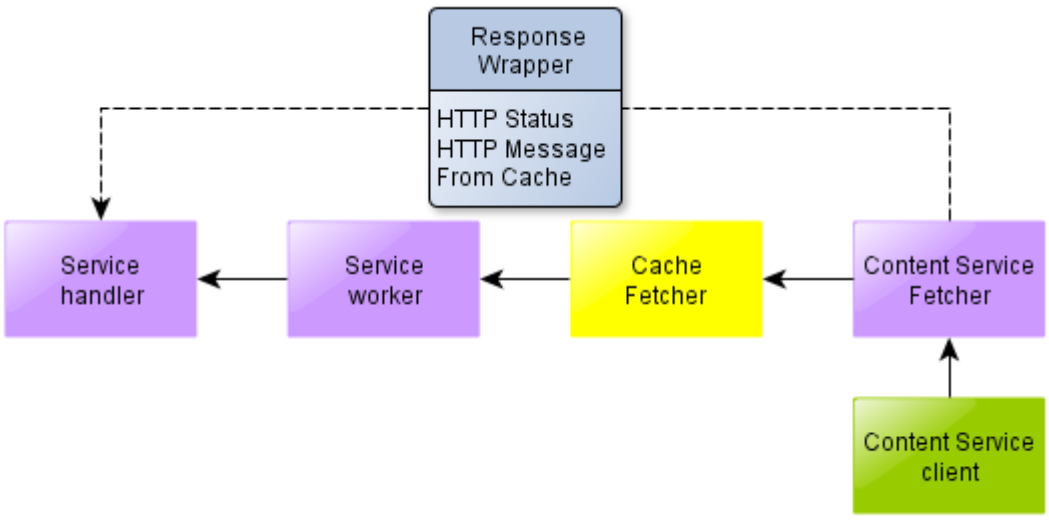

Figure 14. JSON Response Wrapper

Response Wrapper is returned to a Service handler. The Service handler is responsible for providing a CP response. The Service handler uses the HTTP status and the HTTP message to construct the CP response.

Metadata

Fetchers can add or modify metadata in Response Wrapper. Metadata fields can be added. The only metadata field for now is “From Cache”.

Cache Fetcher sets “From Cache” as true when the content is returned from the cache. “From Cache” helps the Worker to delete cached errors and fetch from a CS when an error is returned from the cache.

3.5 Fetcher Implementations

Cache Fetcher

A service can initialize Cache Fetcher instance with “Service name” (for example npc), thread pool settings and database settings. “Service name” identifies service settings and is always provided in the “Get content” procedure.

Cache Fetcher is used for transparent caching. When data is found from the cache, it is given to Request Callback.

Cache Fetcher has synchronous and asynchronous modes. In the asynchronous mode, pool size and other options can be configured.

Cache Fetcher creates a “Fetch Callback” called Cache Callback. Cache Callback stores response content in the cache and then runs Request Callback.

Content will be in Request Callback when content is found from the cache or when running in the synchronous mode. Cache Fetcher will provide content itself when content is found in the cache. When running in the synchronous mode, Cache Fetcher will wait until a Fetcher in the provided “Fetcher chain” places content in Fetcher Callback and returns.

While running in the asynchronous mode a cache miss will probably result in Request Callback not containing response content. It is possible but unlikely that an asynchronous thread places content to Request Callback before the initial thread returns from Cache Fetcher.

When Cache Fetcher is used it will probably always be the first Fetcher in the chain. Caching is a very fast content source and all response content is being cached when Cache Fetcher is always run first.

Cache Fetcher takes the name of the next fetcher in the chain. The service has registered the Fetcher name to the Fetcher factory. Cache Fetcher instantiates the next Fetcher using its name in the Fetcher factory.

Content Service Fetcher

CS Fetcher is a Content Fetcher that contacts the CSP. In a usual service CS Fetcher is invoked by the transparent Cache Fetcher after a cache miss.

The Worker could directly connect the CS Fetcher if there is a reason not to use the cache. This does not necessarily provide product value but it shows that the cache is actually a transparent and modular part of the system.

Asynchronous Fetcher

Currently Cache Fetcher serves asynchronous threading. This design does not make much sense. The design decision was made since Cache Fetcher seemed like the only good place to start asynchronous threads. However, having a separate Fetcher for asynchronous threading would be more flexible. This can be improved during future development.

A requirement for asynchronous threads is to consume as small amount of memory as possible. Asynchronous threads will take time to finish and they may introduce memory leaks. Consuming a small amount of memory in an asynchronous thread greatly reduces the amount of memory that a server consumes.

A threading Fetcher would only be needed in asynchronous fetching. Threading Fetcher could always be right before CS Fetcher. That way only CS Fetcher and recursive Request Callbacks would be in thread memory.

3.6 Transparently Caching Bad Content

Transparent caching presents an issue when bad content is stored in the cache. Caching that is fully transparent and unaware of cached content always returns the cached content as if it was coming from a CSP.

The CSP might have returned an error response for any reason. Even if CSP now returned requested content, the error response would be returned from the cache. Also cached content can be in an invalid format for some reason. Even if CS Fetcher now produced a valid response, the invalid response would be returned from cache.

Resolving this issue requires adding caching decisions to the service. One option is that each Worker refreshes the cache when bad content is returned from the cache. Another option is that CS Fetcher denies caching content when the content is bad. The first option was chosen because the Worker was already aware of caching since it needs to call Cache Fetcher. Also bugs in caching are more robustly resolved when invalid cached content is always refreshed.

Cache Fetcher writes to content metadata that the content is returned from the cache. The Worker checks from content metadata if the content originates from the cache. Content with an invalid format is assumed to originate from the cache. Bad cached content is removed from the cache and retrieved from the CSP.

Caching bad content is not an issue when using the cache for an asynchronous service. Additional Service content is not provided in an authentication response when content is not available. When content becomes available it is provided in the authentication response.

Using the cache simultaneously for caching and asynchronous service presents a problem. The asynchronous service could remember that the content request is being processed by storing an empty value to database. However, in that case inbound service has no way of knowing if cached content is stuck in a bad state or if the request is being asynchronously processed.

This could be solved by tracking content that is actively processed. Instead a design that is easier to implement was chosen. A new asynchronous process is always started when good content is not found in the cache. Content may end up being retrieved multiple times from the CSP.

A more efficient solution can be implemented later. There is currently no service that is asynchronous and needs caching.

3.7 Cache Database Design

Data Structure

The cache data structure needs to be uncoupled from the requirements of individual services. This is achieved by storing data as simple key-data pairs (instead of using a relational data model). Rough drawing of the initial view of the data structure is demonstrated in table 1.

Table 1. Example of an initial data structure

Service	Refresh	Key	Data
ascache	xyz	{"txn":"H16", "msisdn":"+35847001001"}	{"servicex":"y"}
npc	zzz	+35847001001	{"operid":"x", "datasource":"z", status":"y"}

The example stores JSON variable names for all key and data entities as characters. For example the “txn” and “msisdn” variables in the ASCache service key are variable names. The key index handles long character arrays with JSON variables and values. This can be slower than handling only values that are stored in a relational structure.

Minor disadvantages include are redundant JSON variable names in the database and more difficult data mining when JSON is not supported by the database.

The cache has a requirement for efficiently finding deprecated entries. The timestamp column “refresh” tells when the cache was refreshed the last time.

Function-Based Indexes

Function-based indexes can be used to index JSON values [9]. Indexes have to be different for different key formats so services need their own tables. The following is an example key index definition for the asc table:

```
CREATE UNIQUE INDEX key_idx ON asc (asc.key.txn, asc.key.msisdn);
```

The database table name can be configured for each service. Cache manager can construct SQL queries based on a cache query and configuration. Therefore Cache manager can translate the cache query {"**txn**":"H16", "**msisdn**":"+35847001001"} from the **asc** service in to the following: WHERE asc.key.txn='H16' AND asc.key.msisdn='+35847001001'. The database management system should be able to utilize the index key_idx with this query.

Performance is good since an inserted variable will be effectively indexed. A minor downside here is the duplication of key values to the key index.

Cache Data Model

The function-based index approach could not be used in this project since JSON indexing support was released in Oracle 12.1.0.2 [10] and is not present in all required databases.

However, it was decided to use separate tables for different services. The tables will still have an identical structure so that they can be handled in a generalized way. The decision was made for the following reasons:

- Less migration is required if indexing is added later.
- Performance can be more easily improved by simply moving tables to another database.
- Performance is possibly improved by not having to select rows of a specific service and possibly having to enumerate and index service identifiers.
- Debugging is easier when a service can be tracked by monitoring a single table.
- Key and value maximum sizes can be modified depending on service specification.
- Data structure can be easily customized for a service when absolutely required.

Figure 15 shows the columns designed for the final data structure:

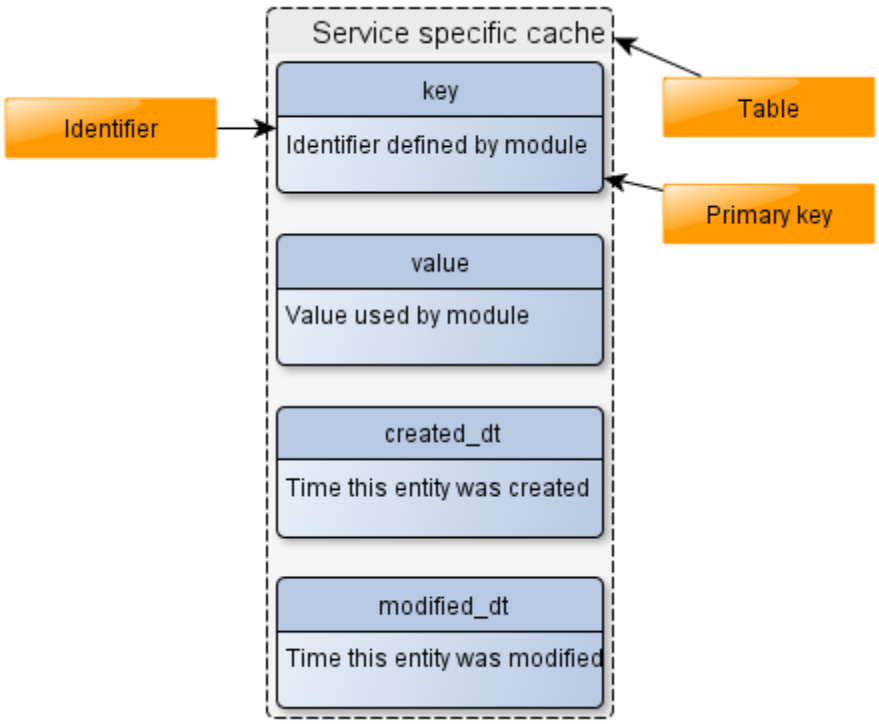

Figure 15. Cache table columns and explanations

One thing to consider is that database functionality of CP should be extendable. For example, statistical or billing information about calls to CSPs could be required. These would be stored in different tables.

Cache Lifecycle

The original NPC does not schedule refresh functionality by itself. It provides an administrative refresh call. Users must schedule the sending of refresh requests on their own. [3,14]

Implementing an internal scheduler to CP requires scheduling configuration and possibly clustering support. Scheduling configuration support is necessary since different caches need to be refreshed at different times or intervals.

When there are multiple CP servers clustered to use the same database, it does not make sense for all of them to run refresh. Of course only one could be configured to run the refresh but it should be possible to use the same configuration on all servers. Kiuru

MSSPs solves this issue by having all servers agree on a master server. The NPC server solves this issue by having an external source that contacts one of the servers.

All necessary functionality is already implemented in Kiuru MSSPs and would only need to be copied to CP. However, this would add a considerable amount of new code, dependencies and documentation to CP.

ASCache has an API call for cleaning the cache. It can be called using an external scheduler process that makes a cache cleaning request. This is similar to the NPC server. ASCache can also be configured to send scheduled refresh requests so there is no need for an external scheduler. This way the customer can decide if they want to use an external scheduler or configure scheduling in ASCache.

The NPC service in CP works in the same way as ASCache. The NPC cache refresh has an API call that can be called from an external scheduler or CP can be configured to call it.

3.8 Cache Database Development

Kiuru Database (DB)

Database configuration and connection instantiation is handled by the Kiuru DB class from the Kiuru core library. Kiuru DB is used to manage database configuration and to manage connection pools. Connections provided by Kiuru DB are used with prepared statements.

Kiuru DB is configured during the server start-up process. The database can be configured with vast amount of options. The most usual are name, URL, username, password, schema, pool settings and driver class. [11,43]

Services can be configured to use any of the configured databases. Multiple services using the same configured database have the same connection pool.

Code Structure

Cache DB is used to perform database operations. Cache DB has methods for inserting, getting and deleting an entry.

Cache DB constructor takes the configured database name and a service name. The constructor saves the database name so it can be used to get connections from Kiuru DB. SQL query table names are derived from the service name. The NPC cache table is, for example, npc_cache.

Multi Database Product Support

Database queries may need to differ in different database products. Therefore supported products have their own Cache DB implementations. Product implementations change SQL query strings or method implementations when generic ones do not work.

A separate instantiating class is used to resolve and instantiate Cache DB implementations. Given a configured database name the instantiating class resolves a database product from Kiuru DB.

Implementation Consideration

Most of the multi database support and code structure was already done well in other Kiuru products. Considerable amount of design and implementation was easily applicable to Cache CB.

There are data schemas and implementations for PostgreSQL and Oracle products. The cache database is so simple that they do not really differ from each other. In the future support for different implementations may become necessary.

Searching the cache with search patterns, or other cache database extensions, may become necessary in the future. Code should support addition of different database operations.

4 Testing

4.1 Testing Methods

NPC Tester

ASCache was tested using a Shell Script library called ShUnit. Methics' previous JSON services were tested with ShUnit library so it is a natural tool to keep using for the NPC service testing.

Any changes in software could cause software bugs. When a tester is run it checks that the software still works as expected. Methics' ShUnit tester sends requests to a CP service and asserts a certain kind of response.

Bash tests are highly portable because of stability and commonness of the Bash environment. Tests should be easy to run in most *nix operating systems. Tests are easier to maintain because many engineers have experience with Bash scripting.

The ShUnit library in itself provides a very limited set of features to help with testing. It is a simple tool for running test tasks that can succeed or fail. ShUnit does not help with request building or response validation. The tester has some issues with understanding JSON but usually this is not a big issue.

The best attribute of ShUnit is that it is really straightforward to run and modify using a Bourne shell. Creating tests with ShUnit, Bourne features and a few common *nix programs is relatively simple.

The worst attribute of ShUnit is unsatisfactory strength of Bourne shell scripting. The Bourne scripting language is not suitable for maintaining large scripts. The Bourne scripting language lacks many basic features that are taken for granted in programming languages. While the Bourne script itself can be considered portable, many of the most powerful programs may not be considered portable. There is lack of any sensible interface and version management with programs.

The Bourne scripting is arguably not a good tool to test services. However, it is used in many of Methics' products. CP works as infrastructure for the NPC service and is not directly subject to automated testing. The NPC interfaces are really simple and ShUnit is certainly capable of testing something as minor as the NPC service. While having a good tester tool would be useful in future CP services, NPC tests were developed in a short time using ShUnit.

Content Service Simulation

Numpac cannot be used for testing so a simulated content service is needed for testing the NPC service. Figure 16 demonstrates the testing with the tester and a simulator:

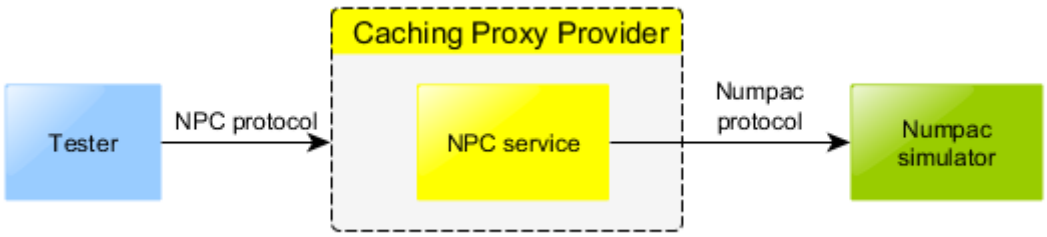

Figure 16. NPC testing with tester and simulator

The tester requests the NPC service for the number portability content of an MSISDN. The simulator provides different responses based on the requested MSISDN.

Initially a technique called chaining was considered to be the best way to provide test data. The test data would have been stored in the cache of an NPC service. With chaining that NPC service would have acted as a content provider to another NPC service.

Methics decided not to implement the chaining support to the NPC service. Providing of the Numpac content had to be simulated. Methics has a simulator product that simulates different services. A Numpac simulator will be a part of the simulator server.

NPC System Testing

The tester cannot test the whole authentication process but only CP and the NPC service. Testing must also be done for the NPC client and the number portability routing in MSSPs.

Setting up an environment for a complete system test requires configuration of an MSSP server. The MSSP can be configured in the following way to test the NPC service:

- The MSSP requests the number portability content from the NPC service.
- The MSSP routes a request based on the NPC service response:
    - An authentication error is returned by default.
    - The authentication request is routed to the mobile network simulator if the tested MSISDN is ported a certain operator.

An authentication request can be made from a command line tool. Authentication for different MSISDNs can be requested to produce different Numpac responses. The MSSP is monitored from the MSSP server log files. This set up is demonstrated in figure 17.

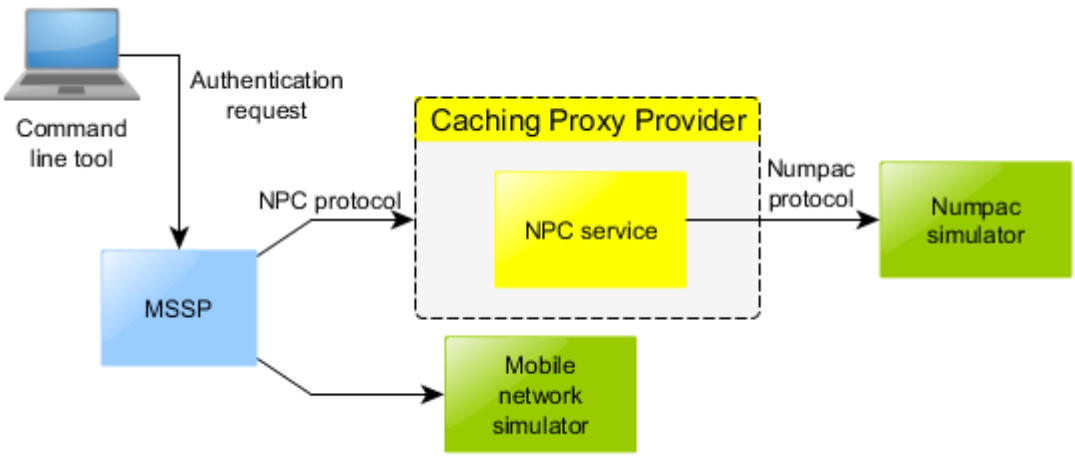

Figure 17. NPC testing with an MSSP and the simulator

4.2 Testing Plan

The NPC system is more difficult to set up than an NPC tester. The NPC tester only needs to be configured with the NPC service location. The NPC system requires configuration in the MSSP that is hard to maintain as part of a development kit.

Constantly testing the NPC in system scope is not a priority. The MSS is being constantly tested and the NPC service has the NPC tester. Without the NPC system, testing the NPC client in an MSSP is the only part that is not well tested.

Running the NPC tester is recommended when:

- Modifying the NPC service or CP.
- Modifying the CP dependencies considerably.
- Modifying the simulated Numpac.
- Modifying the simulator server considerably.

Running a system test is recommended when:

- Modifying the NPC functionality in the MSSP.
- Modifying the MSSP routing code considerably.

5 Deployment

5.1 Scalability and Redundancy

The Caching Proxy cluster is a group of CP servers that are connected to a shared database. Clustered servers usually have the same configuration and are located behind a load balancer.

Data is stored to the cache database and individual CP servers do not have an internal state. Therefore clustered CP servers can be taken offline and only pending connections are affected. Affected connections manifest as exception cases in an MSSP.

Cluster members should not need any direct connection or data sharing between each other. All the relevant data to process Content Service and administration requests should be in the cache database.

Different databases can be synchronized. The CP servers connected to different synchronized databases work as if there were only a single database. This can be used for completely transparent geo redundancy.

5.2 Chaining

In chaining one CP server contacts the next server in chain to see if the next CP server has requested content cached. The next CP server then contacts another CP server in the chain. This continues until content is finally found or requested from a CSP.

The benefit is that there is only one CSP request. All members of the chain cache the response and the response. Later the response is returned from the cache when lower members of the chain request for it.

Multiple CP servers are useful behind a load balancer in which case chaining is an alternative to clustering. Multiple CP servers can also be set in a geo redundant set up in which case chaining is an alternative to database synchronization.

The NPC server has chaining functionality. The NPC server can, instead of contacting a CSP, send a request to another NPC server and handle the response like a CSP response. This is why the NPC server has two requests, a lookup request for the final MSSP response style and a q request for CSP response style.

The NPC server contacts CSP instead when chain fails to respond. Chaining has the benefit of getting a single CSP response stored to multiple caches and finding previously cached responses from higher in chain.

NPC chaining does not have any system in place to populate responses from lower in chain to higher. Therefore when a cache goes offline the previous cache in chain is going to make the request. Caches in the chain starting from the offline cache will not have the CSP response.

Efficiency Consideration

Chaining efficiency could be increased with a system to alter the chain depending on which members of the chain have the best data. Of course chaining always adds useless connections that add delay in case of a cache miss.

There could be a system to spread the cached data to all members of the chain, in which case chaining might or might not be useful. In fact database synchronization would already achieve spreading of cached data in a very flexible and robust manner.

Designing Chaining Service

At first, it was considered that chaining could work like it does in the original NPC. It does not make sense, however, that all services should provide an API request for chaining support. Also NPC chaining handled responses as if they were coming from

the synchronous NPC service provider. Asynchronous chaining would need its own handling.

A general chaining solution that is transparent to the service and client components would have been better as a feature of CP. While chaining, as deemed in efficiency consideration, does not necessarily make sense, it has been a useful challenge in designing. Generalizing the cache design and making caching transparent was the most challenging task of this project.

In the future a design can be challenged by asking whether the service design can work with chaining in the design. As figure 9 shows, the Cache Fetcher should be able to work as a black box. The Worker requires that it is told that no data is found or it gets a response in a format that its CS Fetcher provides. Cache Fetcher could call Chaining Fetcher that would retrieve the response from another server that features the same CS Fetcher.

Given enough routing information Fetchers could potentially form any kind of complex synchronous or asynchronous networks. Of course, there are no such requirements and given that chaining is pretty useless overengineering must be avoided. Generalizing interfaces is not, however, necessarily overengineering. Generalizing interfaces helps to make the cache a transparent component between Workers and CS Fetchers.

The Fetcher design is generalized. Fetchers could be added and removed without breaking the system. Fetchers handle JSON objects so the data could be handed over to another server. While chaining is not implemented it pretty easily could be, proving that the design is generalized.

Alternative to Chaining

There could be a system for asking other servers for cached content. One server could contact a bulk of other servers and ask for cached data.

There would not be long chains that add delay and this could also be performed in parallel. The breaking of chain would not be an issue. Managing connected servers would be easier either with manual configuration or with an automated mesh.

There would be a Fetcher that, after Cache Fetcher, would contact other CP servers. The other servers would have a service for cache requests. The service would take the service name and the key of the content being requested and retrieve the content from the database.

The Fetcher would continue to CS Fetcher if the content was not found from the other CP servers. The Fetcher might distribute the response content to other servers when CS Fetcher returns.

Implementing this alternative method is not necessary for now. Implementation is unnecessary for the same reasons that chaining is unnecessary.

6 Results

MSSP Operation

The provided content is identified with a “Fetch key”. The service regulates the scope of caching by defining a key format.

Services that are used for Additional Services content are asynchronous. A service can configure Cache Fetcher for asynchronous processing.

Defining a transaction specific key format makes caching useful only for temporary asynchronous storage. Defining a less detailed key format causes cached content to be used for caching.

The NPC service provides a call for cleaning a cache entry. This way the cached content can be cleaned in case of a routing error. The NPC service provides MSSPs the content that NPC fetches from Numpac. The fetched content is, in a priority order, one of the following: the operator of a ported number, the operator based on a number prefix or an unknown operator.

Service Deployment

CP was designed to be scalable. Servers can be placed in a load balanced set up. In the load balance set up multiple servers use the same cache database. Caching logic expects that other servers use the same database and it works well in this situation.

A geographically redundant set up is possible. Databases between geographical locations can be configured to synchronize with each other. In this set up CPPs do not need to know about geographical redundancy.

CP Development

Workers can add and remove Fetchers easily. Workers may still need some Fetcher specific logic such as handling cached error content. Fetchers can add their own metadata to communicate with Workers.

Fetchers can be easily created. A Fetcher needs to:

- Implement the Content Fetcher interface.
- Contact the next Fetcher in chain when content is not found.
- Return a response to callback.

CP has only the NPC service for now. Different external and internal services can benefit from CP in the future.

7 Conclusions

CP provides proxy services to the Kiuru MSSP servers. Kiuru MSSP and CP are products developed by Methics. The purpose, functionality and development of CP are documented in this thesis.

Use Cases

Kiuru CP eases the work of the MSSP servers by providing asynchronous services for Additional Services content and caching services for routing content. CP requirements are based on these two known use cases.

CP has several extension points for new possible use cases. New Fetchers are easy to create and communication between Fetchers is done with generic JSON datatypes.

Extensible design of CP helps with adding new use cases. Any communication in an MSSP system could potentially go through a CPP. The design is still based on routing and Additional Service use cases. New use cases may require improving or changing the CP design.

Fetcher Design

The design of Fetchers and transparent caching may have been too sophisticated. Instead of using the “Fetcher chain”, Workers could separately check if data is in cache and then proceed to request for it.

The fetcher design is based on an arguable aim that caching should be transparent to a service. However, complete transparency is often not possible. For example, the NPC cache refresh API call is entirely based on caching. Therefore caching cannot possibly be transparent to NPC service.

Usage of JSON typed data works when it is logged well. JSON data can become confusing when it is passed through multiple layers of software and its state in different layers is not logged.

NPC Service

The NPC service works as a result of the project. The NPC service has a set of tests for checking that future changes to the software do not break the NPC service.

Testing is especially important since the CP code may change when new services are added. The CP also has dependencies on the Kiuru platform. Changes in the Kiuru platform or its libraries could break the NPC service.

Future Development

More CP services will be added to CP in the future. It is unlikely that more content sources (in addition to the cache) are added.

Asynchronous services have not been tested on CP since there are no service implementations. Some bugs will probably appear, especially in Cache Fetcher, when asynchronous services are added.

Thesis Future Value

This thesis documents how CP was developed to provide content in optimized, generalized, scalable and secure manner. Also the consideration of multiple use cases and future extensibility is in a central role.

Hopefully this thesis helps other software developers that are developing software with similar requirements. Especially developers of software that provides proxy services and requires a generalized design can benefit from this thesis.

References

1 European Telecommunications Standards Institute (ETSI). Mobile Commerce (M-COMM); Mobile Signature Service; Web Service Interface TS 102 204 V1.1.4 [online]. Sophia Antipolis Cedex: ETSI; August 2003.
URL: http://docbox.etsi.org/EC_Files/EC_Files/ts_102204v010104p.pdf. Accessed 24 February 2015.

2 JSON [online].
URL: http://www.json.org. Accessed 16 January 2015.

3 Methics. Kiuru Number Portability Cache 1.0 User Guide version 1.2. Helsinki: Methics; 2013.

4 European Telecommunications Standards Institute (ETSI). Mobile Commerce (M-COMM); Mobile Signature Service; Specifications for Roaming in Mobile Signature Services TS 102 207 V1.1.3 [online]. Sophia Antipolis Cedex: ETSI; August 2003.
URL: http://docbox.etsi.org/EC_Files/EC_Files/ts_102207v010103p.pdf. Accessed 24 February 2015.

5 Federation of Finnish Financial Services (FFI). Banks' Tupas Certification Service for Service Providers version 2.1. FFI; 2005.

6 Internet Engineering Task Force (IETF). The WebSocket Protocol RFC 6455 [online]. IETF; December 2011.
URL: https://tools.ietf.org/html/rfc6455. Accessed 3 March 2015.

7 Accenture. Kyselypalvelun tekninen kuvaus (Numpac technical description) 1.4. Suomen Numerot Numpac Oy; 2012.

8 Methics. Kiuru Additional Context Design Specification version 0.8. Helsinki: Methics; 2014.

9 Oracle. JSON in Oracle Database [online]. Oracle.
URL: http://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6397. Accessed 16 January 2015.

10 Oracle. Oracle Database 12c Release 1 (12.1.0.2) New Features [online]. Oracle.
URL:
http://docs.oracle.com/database/121/NEWFT/chapter12102.htm#BGBGADCC.
Accessed 16 January 2015.

11 Methics. Kiuru MSSP 5.0 Configuration Guide version 1.6. Helsinki: Methics; 2015.